Smart Pointer in C++

Presenter: Haotao Lai (Eric)
Contact: haotao.lai@gmail.com

Table of Content

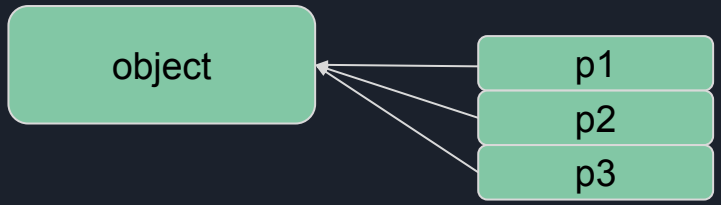
What Smart Pointer can do ?

Basically, solve two problems:

- Dangling Pointer Problem
- Memory Leak Problem

Dangling Pointer

Several pointer-variables point the the same object, at some point, the object get deleted via one of the pointer, but the rest don't know, keep manipulating the object.



Example code:

```
delete p3;
// do something with p1 or p2
```

Memory Leak

You allocate a piece of memory, but for some reasons, you never release it.

Example code:

```
int foo( ) {
    int *i_ptr = new int(1);
    If ( *i_ptr == 1 ) {
        // memory leak occurs, forget to delete
        return 2;
    }
    delete i_ptr;
    Return 0;
}
```

Solution is Smart Pointer

Requirements

Smart Pointer comes into C++ in C++11 (or known as C++0x). The auto_ptr was introduced since C++98 but now it is discarded.

Want to use smart pointer, you need to include a header file named “memory”:

```
#include <memory>
```

Also for older version compiler, you may need to specify the flag:

“-std=c++11” or “-std=c++0x”

What is Smart Pointer

An object that overloads the pointer-related operator which allows you to treat this object as a pointer (syntax equivalence). Remember, smart pointer is an object, not a “real raw pointer”.

The main purpose is to free the programmer from being worried about the dynamic allocated memory.

Point.h

```
#include <iostream>

using std::cout;
using std::endl;

class Point {
public:
Point(int x=0, int y=0) : x(x), y(y) {
  cout << "Point(" << x << ", " << y << ") is created." << endl;
}
~Point() {
  cout << "Point(" << x << ", " << y << ") is destroyed." << endl;
}
void toString() {
  cout << "Point(" << x << ", " << y << ")" << endl;
}
private:
    int x, y;
};
```

auto_ptr

Already be discarded, never try to use it any more.

Take an example, the following program will crash.

```
int funcError(auto_ptr<Point> p) {
    // do something with p
}

int main() {
    auto_ptr<Point> p( new Point(1, 1) );
    funcError(p);
    p->toString(); // program will crash here
    return 0;
}
```

Why crash?

Since the ownership of p has been transferred to funcError and the pointer got deleted when it ran out of scope after funcError finish executing

auto_ptr

Already be discarded, never try to use it any more.

Other limitations:

- Cannot point to an array
- Cannot work with container

unique_ptr

You can imagine “unique_ptr” is designed to substitute the “auto_ptr”. It always guarantee that at anytime one resource will only be pointed by one unique_ptr pointer. Once the pointer-variable runs out of scope the resource will be deallocate.

Example usages:

```
unique_ptr<int> uptr( new int );
unique_ptr<int[ ]> uptr( new int[5] );
```

unique_ptr doesn’t support assignment and copy semantics, it only has “move semantic”. Also, there is no pointer increment or decrement operation with unique_ptr.

unique_ptr

An wrong example:

```
int func(unique_ptr<Point> p) {
    // do something with p
}

int main() {
    unique_ptr<Point> p( new Point(1, 1) );

    // the following line is
    // invalid, cannot be copied
    func(p);

    return 0;
}
```

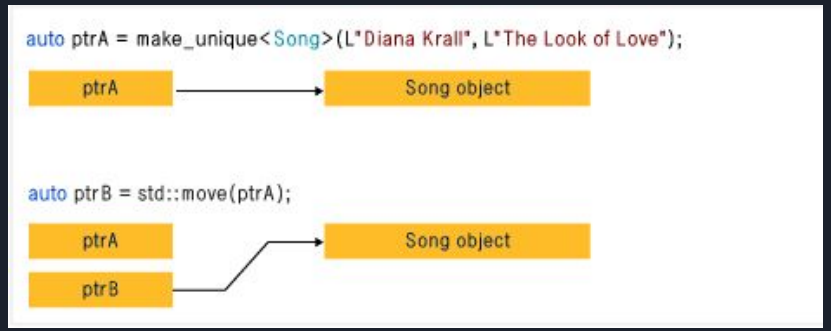
An correct example:

```
int func(unique_ptr<Point>& p) {
    // do something with p
}

int main() {
    unique_ptr<Point> p( new Point(1, 1) );
    func(p);
    return 0;
}
```

unique_ptr

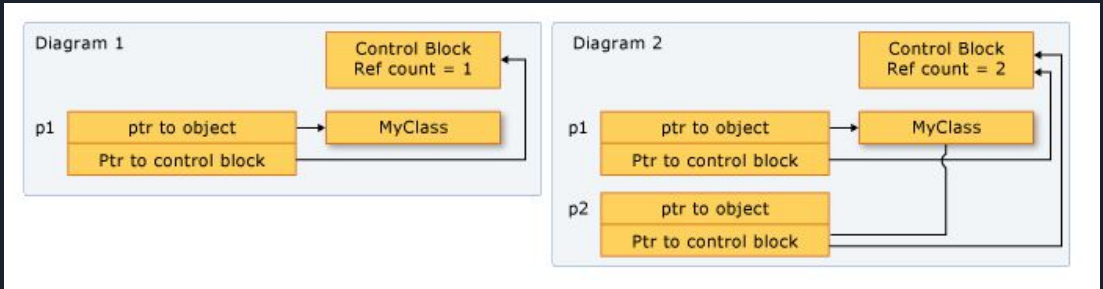
It cannot be copied, but can be moved (to transfer) the ownership (of the object).



shared_ptr

shared_ptr is a smart pointer that retains shared ownership of an object through a pointer. Several shared_ptr objects may own the same object.

The object will get deleted when the “ref count” go down to 0 inside the control block.



shared_ptr

How reference counter work?

- When a new shared_ptr point to the same object, counter plus one
- When a copy of the shared_ptr happens, counter plus one
- When a shared_ptr variable out of scope, counter minus one

shared_ptr

Example usage

```
#include <memory>
#include <iostream>

using std::cout;
using std::endl;
using std::shared_ptr;

int fun(shared_ptr<int> ptr) {
    cout << "[fun] object 100 reference count: " << ptr.use_count() << endl;
    return 0;
}
shared_ptr<int> create() {
    shared_ptr<int> ptr(new int(50));
    cout << "[create] object 50 reference count: " << ptr.use_count() << endl;
    return ptr;
}
int main(int argc, char const *argv[]) {
    shared_ptr<int> ptr1(new int(100));
    shared_ptr<int> ptr2 = ptr1;
    fun(ptr1);
    cout << "[main] object 100 reference count: " << ptr1.use_count() << endl;
    shared_ptr<int> ptr3 = create();
    cout << "[main] object 50 reference count: " << ptr3.use_count() << endl;
    return 0;
}
```

shared_ptr

I will say shared_ptr is the most common use smart pointer in reality development. But there will be a problem when circular dependencies occurs. If that is the case, the reference count will never go down to 0 which means the pointing object will never be deleted.

```
class A {
public:
    shared_ptr<B> m_b;
};

class B {
public:
    shared_ptr<A> m_a;
};
```

weak_ptr

weak_ptr models temporary ownership: when an object needs to be accessed only if it exists, and it may be deleted at any time by someone else, weak_ptr is used to track the object, and it is converted to shared_ptr to assume temporary ownership. If the original shared_ptr is destroyed at this time, the object's lifetime is extended until the temporary std::shared_ptr is destroyed as well.

In addition, weak_ptr is used to break circular references of shared_ptr.

weak_ptr

```cpp
#include <iostream>
#include <memory>

std::weak_ptr<int> gw;

void observe() {
    std::cout << "use_count == " << gw.use_count() << ": ";
    if (auto spt = gw.lock()) { // Has to be copied into a shared_ptr before usage
        std::cout << *spt << "\n";
    }
    else {
        std::cout << "gw is expired\n";
    }
}

int main() {
    {
        auto sp = std::make_shared<int>(42);
        gw = sp;

        observe();
    }
    observe();
}
```

Output:

```
use_count == 1: 42
use_count == 0: gw is expired
```